第 8 章　Java 的多线程机制

支持多线程是现代操作系统的一大特点。Java 语言是跨平台的，无法像 C/C++语言一样通过调用系统的 API 来实现多线程程序，所以它在语言本身加入了对多线程的支持。而且，所有的多线程功能都是以面向对象的方式来实现，学习起来比较简单，控制也很方便。本章就将对 Java 的多线程机制做一个简明的介绍。

8.1　线程的概念

在操作系统中，通常将进程看作是系统资源的分配单位和独立运行的基本单位。一个任务就是一个进程。比如，现在正在运行 IE 浏览器，同时还可以打开记事本，系统就会产生两个进程。通俗地说，一个进程既包括了它要执行的指令，也包括了执行指令时所需要的各种系统资源，如 CPU、内存、输入输出端口等。不同进程所占用的系统资源相对独立。进程具有动态性、并发性、独立性和异步性等特点。

一般情况下，程序员并不需要对进程有更多的了解。本书前面编写的程序一旦被执行，都是独立的进程。它所需要的资源，大多数由操作系统来自动分配，无须程序员操心。

线程是一个比较新的概念，在 20 世纪 80 年代末才正式被引入。它在提高系统吞吐率、有效利用系统资源、改善用户之间的通讯效率以及发挥多处理机的硬件性能等方面都有显著的作用。因此，线程在现代操作系统中得到了广泛的应用，如 Windows、Unix、Linux 等都提供了多线程机制。

线程是比进程更小的执行单位。某一个进程在执行过程中，可以产生多个线程。每个线程都有自己相对独立的资源（这个和进程非常相似）、生存周期。线程之间可以共享代码和数据、实时通信、进行必要的同步操作。

在一个进程中，可以有一个或多个线程的存在。如果程序员不创建线程对象，那么系统至少会创建一个主线程。

8.1.1　多线程的特点

在基于线程（thread-based）的多任务处理环境中，线程是最小的执行单位。这意味着一个程序可以同时执行两个或者多个任务的功能。例如，一个文本编辑器可以在打印的同时格式化文本。所以，多进程程序处理“大图片”，而多线程程序处理细节问题。多线程程序比多进程程序需要更少的管理费用。进程是重量级的任务，需要分配它们自己独立的地址空间，进程间通信是昂贵和受限的，进程间的转换也是很需要花费的。但是，线程是

轻量级的选手，它们共享相同的地址空间并且共同分享同一个进程。线程间通信是便宜的，线程间的转换也是低成本的。当 Java 程序使用多进程任务处理环境时，多进程程序不受 Java 的控制，而多线程则受 Java 控制。

设计好的多线程，能够帮助程序员写出 CPU 最大利用率的高效程序，因为 CPU 的空闲时间保持最低。这对 Java 运行的交互式的网络互连环境是至关重要的，因为空闲时间是公共的。举个例子来说，网络的数据传输速率远低于计算机的处理能力，本地文件系统资源的读写速度远低于 CPU 的处理能力，当然，用户输入也比计算机慢很多。在传统的单线程环境中，你的程序必须等待每一个这样的任务完成后，才能执行下一步——尽管 CPU 有很多空闲时间。多线程使你能够获得并充分利用这些空闲时间。

进程和线程最大的区别在于：进程是由操作系统来控制的，而线程是由进程来控制的。所以很多由操作系统完成的工作必须交由程序员完成。前面所写的程序都是单线程的程序，如果需要设计多线程的程序，难度就要大一些。进程都是相互独立，各自享有各自的内存空间。而一个进程中的多个线程是共享内存空间的，这意味着它们可以访问相同的变量和对象，这一方面方便了线程之间的通讯，另一方面又带来了新的问题：多个线程同时访问一个变量可能会出现意想不到的错误。

在传统的 C/C++、OP 等语言中，都是利用操作系统的多线程支持库来完成多线程的程序设计，线程之间的同步、异步、并发、互斥等控制起来比较麻烦（当然好的开发环境也会用类来对这些进行封装）。而 Java 在语言这一级提供了对多线程的支持，它本身就提供了同步机制，大大方便了用户，降低了设计程序的难度。

编制多线程程序，对于程序员而言是一个极大的挑战。尽管 Java 的线程类已经做得不错了，但还远称不上完美。如果需要编制大型的、要求可靠性很高的多线程程序，还需要程序员花费大量的时间来设计和调试。如果要深入介绍线程控制的每一个细节，足够写出厚厚的一本书。限于篇幅，本章只做一些简明介绍。

8.1.2　线程的状态

Java 中用 Thread 类和它的子类对象来表示线程。一个线程总是处于下面 5 种状态之一：

- 新建：当创建一个 Thread 类和它的子类对象后，新产生的线程对象处于新建状态，并获得除 CPU 外所需的资源。
- 就绪：当处于新建状态的线程被启动后，将进入线程队列等待 CPU 资源。这时，它已经具备了运行的条件，一旦获得 CPU 资源，就可以脱离创建它的主线程独立运行了。另外，原来处于阻塞状态的线程结束阻塞状态后，也将进入就绪状态。
- 运行：当一个就绪状态的线程获得 CPU 时，就进入了运行状态。每个 Thread 类及其子类对象都有一个 run()方法，一旦线程开始运行，就会自动运行该方法。在 run()方法中定义了线程所有的操作。
- 阻塞：一个正在运行的线程因为某种特殊的情况，比如，某种资源无法满足，会让出 CPU 并暂时停止自身的运行，进入阻塞状态。只有当引起阻塞的原因消除时，它才能重新进入就绪状态。

- 死亡：不具备继续运行能力的线程处于死亡状态。这一般是由两种情况引起的：一种是 run()方法已经运行完毕了，另一种是由其他的线程（一般是主线程）强制终止它。

需要指出的是：处于就绪状态的线程是在就绪队列中等待 CPU 资源的，而一般情况下，就绪队列中会有多个线程。为此，系统会给每一个线程分配一个优先级，优先级高的可以排在较前面的位置，能优先得到 CPU 资源。对于优先级相同的线程，一般按照先来先服务的原则调度。

8.2 Thread 类

在 Java 中，有两种方法可以创建线程：一种是继承 Thread 类；另一种是实现 Runnable 接口。但不管采用哪种方式，都要用到 Java 类库中的 Thread 类以及相关方法。

8.2.1 Thread 类的构造方法

Thread 类的构造方法有多个，各有各的用途，如表 8.1 所示。

表 8.1 Thread类的构造方法

构 造 方 法	说 明
Thread()	构造一个线程对象
Thread(Runnable target)	构造一个线程对象，target是被创建线程的目标对象，它实现了Runnable接口中的run()方法
Thread(String name)	用指定字符串为名构造一个线程对象
Thread(ThreadGroup group, Runnable target)	在指定线程组中构造一个线程对象，使用目标对象target的run()方法
Thread(Runnable target, String name)	用指定字符串为名构造一个线程对象，使用目标对象target的run()方法
Thread(ThreadGroup group, Runnable target, String name)	在指定线程组中构造一个线程对象，以name作为它的名字，使用目标对象target的run()方法
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	在指定线程组中构造一个线程对象，以name作为它的名字，使用目标对象target的run()方法，stackSize指定堆栈大小。

表中参数 Runnable 是一个接口，将在 8.3.2 小节介绍。

8.2.2 Thread 类的常用方法

为了能让线程正常运行以及方便程序员对线程的控制，Thread 类提供了很多辅助方法。其中，比较常用的方法如表 8.2 所示。

表 8.2　Thread的常用方法

方　法　名	说　　明
static int activeCount()	返回线程组中正在运行的线程数目
void checkAccess()	确定当前运行的线程是否有权限修改线程
static Thread currentThread()	判断当前哪个线程正在执行
void destroy()	销毁线程，但不收回资源
static void dumpStack()	显示当前线程中的堆栈信息
static int enumerate(Thread[] tarray)	将当前线程组中的线程复制到数组tarray中
String getName()	返回线程的名字
int getPriority()	获取线程的优先级
ThreadGroup getThreadGroup()	获取线程所属的线程组
static boolean holdsLock(Object obj)	当前线程被观测者锁定时，返回真
void interrupt()	中断线程
static boolean interrupted()	测试当前线程是否被中断
boolean isAlive()	测试线程是否已经正常活动
boolean isDaemon()	测试线程是否在后台
boolean isInterrupted()	测试本线程是否被中断
void join()	等待，直到线程死亡
void join(long millis)	等待线程死亡，但最多只等待millis毫秒
void run()	如果类是使用单独的Runnable对象构造的，将调用Runnable对象的run方法，否则本方法不做任何事情就返回了。如果是子类继承Thread类，请务必实现本方法以覆盖父类的run方法
void setDaemon(boolean on)	将线程标记为后台或者用户线程
void setName(String name)	设置线程的名字为name
void setPriority(int newPriority)	改变线程的优先级，Java定义了三种级别：Thread.MIN_PRIORITY、Thread.MAX_PRIORITY 、 和Thread.NORM_PRIORITY
static void sleep(long millis)	正在运行的线程睡眠（暂停），参数millis指定毫秒数
static void sleep(long millis, int nanos)	正在运行的线程睡眠（暂停），millis指定毫秒数，nanos指定纳秒数
void start()	启动线程，JVM会自动调用run()方法
static void yield()	正在运行的线程暂停，同时允许其他的线程运行

8.3　多线程程序的编写

8.2 节介绍了 Thread 类，但是如果要编写多线程程序，是无法直接使用该类的。用户需要继承 Thread 类或实现 Runable 接口才行。无论采用哪一种方法，程序员要做的关键性操作有三个：

- 定义用户线程的操作，也就是定义用户线程的 run()方法。
- 在适当的时候建立用户线程实例，也就是用 new 来创建对象。
- 启动线程，也就是调用线程对象的 start()方法。

下面通过几个例子分别来介绍这两种方式实现的多线程程序。

8.3.1 利用 Thread 的子类创建线程

要创建一个多线程程序，首先要写一个子类继承 Thread 类，并覆盖其中的 run()方法。run()方法中的代码就是这个线程要实现的功能。然后再创建子类对象，这和创建普通类的对象是一样的。最后调用 start()方法启动线程。如果要对线程加以其他的控制，就需要使用 Thread 类的其他辅助方法。

【例 8.1】 用 Thread 子类创建多线程程序。

先定义一个 Thread 的子类，该类的 run 方法只用来输出一些信息。

//----------文件名 myThread.java，程序编号 8.1----------

```
public class myThread extends Thread{//定义 Thread 类的子类
  private static int count=0;   //这是静态变量，所有线程对象共享
  //覆盖 run 方法，实现自己的功能
  public void run(){
    int i;
    for(i=0;i<100;i++){
      count = count+1;
      System.out.println("My name is "+getName()+" count= "+count);
      try{
         sleep(10);              //休眠 10 毫秒，让其他线程有机会运行
      }catch (InterruptedException e) { }
   }
  }
  public myThread(String name){
    super(name);
  }
}
```

下面这个程序用来创建线程对象并运行该线程对象。

//----------文件名 mulThread.java，程序编号 8.2----------

```
public class mulThread{
  public static void main(String argv[]){
    myThread trFirst,trSecond;
    //创建两个线程对象
    trFirst=new myThread("First Thread");
    trSecond=new myThread("Second Thread");
    //启动这两个线程
    trFirst.start();
    trSecond.start();
  }
}
```

类中的成员 count 是一个静态变量，两个线程对象会共享这个变量。每个线程都会将这个变量循环加上 100 次，使得它的值最终变成 200。

按照程序员的设想，当程序运行时，第一个线程先运行，将 count 的值加 1，而后输出，再转入休眠状态；而后第二个线程按照同样的方法运行；如此交替运行，重复进行 100 次。程序的输出似乎也验证了这一点：

```
My name is First Thread count= 1
My name is Second Thread count= 2
My name is First Thread count= 3
My name is Second Thread count= 4
My name is First Thread count= 5
My name is Second Thread count= 6
……
```

但问题远不是这么简单，当程序接着运行下去时，读者可能会看到这样的输出：

```
My name is Second Thread count= 152
My name is First Thread count= 153
My name is First Thread count= 154
My name is Second Thread count= 155
My name is First Thread count= 156
```

其中，First Thread 线程出现了连续两次运行的情况。也就是说，当线程调度时，系统不能够保证各个线程会严格交替地运行，它的调度具有一定的“随意性”。即当两个优先级相同的线程都进入就绪态后，调用哪一个都是有可能的，因此程序的输出结果不可预测。

最为极端的情况下，可能会出现下面这样的输出：

```
My name is Second Thread count= 172
My name is First Thread count= 172
My name is First Thread count= 173
My name is Second Thread count= 174
```

最后 count 的结果小于 200。读者可能会觉得这不可思议，但的确有可能发生。设想下面的情况：

（1）线程 2 取得 count 的值（假定为 171），但还没执行完 count=count+1 这条语句（这条语句实际上是由取得 count 值、将 count 值加 1 和将新值赋给 count 等三条指令组成），线程 1 取得 CPU 的运行权，线程 2 进入就绪队列。

（2）线程 1 取得 count 的值（此时仍为 171），然后执行 count=count+1，现在 count 变成 172。

（3）线程 2 取得 CPU 的运行权，线程 1 进入就绪队列。

（4）线程 2 执行 count+1 和赋值指令，由于它前面取得的值是 171，所以 count 变成 172。

（5）线程 2 执行输出语句，输出 count = 172，然后自动转入休眠状态。

（6）线程 1 取得运行权，输出 count = 172，然后转入休眠状态。

上面的过程完全符合逻辑，但显然共享变量 count 的值不符合预期。所以，当线程共

享某个变量或资源时，一定要做好变量或资源的保护工作。关于如何做到这一点，将在 8.4.1 小节中介绍。

8.3.2 实现 Runnable 接口创建线程

除了继承 Thread 类，还有一种方式可以实现多线程程序：实现 Runnable 接口。Runnable 接口中只有一个方法：run()，因此只要实现了该方法，就可以写成多线程的形式。但问题是，Thread 所拥有的其他辅助方法都不存在，如果这些方法全部由程序员来实现，就过于繁琐。

为了解决这一问题，在实际编程中，实现 Runnable 的子类中，通常都会定义一个 Thread 类的对象，然后利用 Thread 的构造方法：

```
Thread(Runnable target) 或 Thread(Runnable target, String name)
```

将本类作为参数传递给 Thread 对象，这样就可以指定要运行的 run()方法。同时，还可以使用 Thread 类中定义好的其他辅助方法。

除此之外，为了启动这个线程，还需要定义一个 start()方法，以启动内部的 Thread 对象。

【例 8.2】 继承 Runnable 接口实现多线程。

//----------文件名 ThreadImRunnable.java，程序编号 8.3----------

```
public class ThreadImRunnable implements Runnable{
  private static int count=0;
  private Thread trval; //需要一个 Thread 对象
  //定义自己的 run()方法
  public void run(){
    int i;
    for(i=0;i<100;i++)
    { count++;
      //使用 Thread 对象的方法
      System.out.println("My name is "+trval.getName()+" count= "+count);
      try{
        trval.sleep(10);
      }catch (InterruptedException e) {  }
    }
  }
  //实现构造方法
  public ThreadImRunnable(String name){
    trval=new Thread(this,name);  //将本对象传递给 trval 对象
  }
  //定义自己的 start()方法来启动 trval 对象的线程
  public void start(){
    trval.start();
  }
}
```

主程序和前面的相同：

//----------文件名 mulThread.java，程序编号 8.4----------

```
public class mulThread{
  public static void main(String argv[]){
    ThreadImRunnable trFirst,trSecond;
    trFirst=new ThreadImRunnable("First Thread");
    trSecond=new ThreadImRunnable("Second Thread");
    trFirst.start();
    trSecond.start();
  }
}
```

程序 8.3 和前面的程序 8.1 实现同样的功能，不过程序 8.3 明显要比程序 8.1 麻烦一些。那么，到底在什么时候使用 Thread 类，在什么时候使用 Runnable 接口呢？由于 Java 不支持多重继承，所以当某类已经是某个类的子类，而同时又要完成多线程任务时，就可以考虑实现 Runnable 接口。典型的例子是 Applet 程序，由于所有的 Applet 程序都必须是 Applet 的子类，所以如果要实现多线程任务，只能通过实现 Runnable 接口。

8.3.3　使用 isAlive()和 join()等待子线程结束

在 8.3.2 节的例 8.1 中，主程序 main()其实也是一个线程，它也会有结束的时候。那么它到底是等待子线程结束之后才结束，还是自己先结束了呢？所以在程序 8.2 中加入一条输出语句：

//----------文件名 mulThread.java，程序编号 8.5----------

```
public class mulThread{
  public static void main(String argv[]){
    myThread trFirst,trSecond;
    //创建两个线程对象
    trFirst=new myThread("First Thread");
    trSecond=new myThread("Second Thread");
    //启动这两个线程
    trFirst.start();
    trSecond.start();
    System.out.println("主线程结束");
  }
}
```

它的输出结果如下：

```
主线程结束
My name is First Thread count= 1
My name is Second Thread count= 2
My name is First Thread count= 3
My name is Second Thread count= 4
……
```

从输出结果中可以看出，主线程在启动两个线程之后就自行结束了。但在大多数情况

下，用户希望主线程最后结束，这样可以做一些扫尾工作。一种简单的方法是通过在 main()中调用 sleep()来实现，经过足够长时间的延迟以确保所有子线程都先于主线程结束。然而，这不是一个令人满意的解决方法，它也带来一个大问题：一个线程如何知道另一线程已经结束？幸运的是，Thread 类提供了回答此问题的方法。

有两种方法可以判定一个线程是否结束。第一，可以在线程中调用 isAlive()。这种方法由 Thread 定义，它的通常形式如下：

```
final boolean isAlive()
```

如果所调用线程仍在运行，isAlive()方法返回 true，如果不是，则返回 false。但 isAlive()很少用到，因为它需要用一个循环来判断，这样太耗 CPU 资源。等待线程结束更常用的方法是调用 join()，描述如下：

```
final void join()throws InterruptedException
```

该方法名字来自于要求线程等待直到指定线程参与的概念。join()的附加形式允许给等待指定线程结束定义一个最大时间。

下面是程序 8.5 的改进版本，运用 join()以确保主线程最后结束。

【例 8.3】 join()方法使用示例。

//----------文件名 demoJoin.java，程序编号 8.6----------

```
public class demoJoin{
  public static void main(String argv[]){
    myThread trFirst,trSecond;
    //创建两个线程对象
    trFirst=new myThread("First Thread");
    trSecond=new myThread("Second Thread");
    //启动这两个线程并等待它们结束
    try{
      trFirst.start();
      trSecond.start();
      trFirst.join();
      trSecond.join();
    }catch(InterruptedException e){
      System.out.println("主线程被中断");
    }
    System.out.println("主线程结束");
  }
}
```

程序的输出结果如下：

```
……
My name is First Thread count= 197
My name is Second Thread count= 198
My name is First Thread count= 199
My name is Second Thread count= 200
主线程结束
```

使用 join()方法很好地完成了预想的任务。

8.3.4　设置线程优先级

默认情况下，所有的线程都按照正常的优先级来运行及分配 CPU 资源。JVM 允许程序员自行设置线程优先级。理论上，优先级高的线程比优先级低的线程获得更多的 CPU 时间。实际上，线程获得的 CPU 时间通常由包括优先级在内的多个因素决定（例如，一个实行多任务处理的操作系统如何更有效的利用 CPU 时间）。

一个优先级高的线程自然比优先级低的线程优先。举例来说，当低优先级线程正在运行，而一个高优先级的线程被恢复（例如，从沉睡中或等待 I/O 中），它将抢占低优先级线程所使用的 CPU。理论上，等优先级线程有同等的权力使用 CPU。但由于 Java 是被设计成能在很多环境下工作的，一些环境下实现多任务处理从本质上与其他环境不同。为安全起见，等优先级线程偶尔也受控制。这保证了所有线程，在无优先级的操作系统下都有机会运行。实际上，在无优先级的环境下，多数线程仍然有机会运行，因为很多线程不可避免地会遭遇阻塞，例如，等待输入输出。遇到这种情形，阻塞的线程被挂起，其他线程运行。

设置线程的优先级，需要用到 setPriority()方法，该方法也是 Thread 的成员。它的通常形式为：

```
final void setPriority(int level)
```

其中，level 指定了对所调用线程的新的优先权的设置。Level 的值必须在 MIN_PRIORITY～MAX_PRIORITY 范围内。通常，它们的值分别是 1 和 10。默认值是指定 NORM_PRIORITY，该值为 5。这些优先级在 Thread 中都被定义为 final 型常量。

用户也可以通过调用 Thread 的 getPriority()方法来获得当前的优先级设置。该方法如下：

```
final int getPriority()
```

例 8.4 阐述了两个不同优先级的线程，运行于具有优先权的平台，这与运行于无优先级的平台不同。一个线程设置了高于普通优先级两级的级数，另一线程设置的优先级则低于普通级两级。两线程被启动并允许运行 10 秒。每个线程执行一个循环，记录反复的次数。1 秒后，主线程终止了两线程。然后显示两个线程循环的次数。

【例 8.4】 设置线程优先级示例。

//----------文件名 clicker.java，程序编号 8.7----------

```
public class clicker extends Thread{
  private int click = 0;
  private volatile boolean running=true; //循环控制变量
  public int getClick(){
    return click;
  }
  public void run(){
    while (running)
```

```
    click = click + 1;//计数器加 1
  }
  public void normalStop(){
    running = false;
  }
}
```

程序中的循环变量 running 被声明成 volatile，这个关键字告诉编译器，不要自作主张为它进行编译优化。

还有一点，注意不要将循环体中的“click=click+1”改成“++click”的形式。对于前者，编译器会生成多条指令，执行过程中系统有机会将它中断。而后者只有一条指令，系统不能将其中断，这样其他的线程就难以有机会获得 CPU。

//----------文件名 demoPri.java，程序编号 8.8----------

```
public class demoPri{
  public static void main(String argv[]){
    clicker trHigh,trLow;
    //创建两个线程对象
    trHigh=new clicker();
    trLow=new clicker();
    //分别设置优先级
    trHigh.setPriority(Thread.NORM_PRIORITY+2);
    trLow.setPriority(Thread.NORM_PRIORITY-2);
    //启动这两个线程
    trLow.start();
    trHigh.start();
    try{
      Thread.sleep(1000); //等待 1 秒钟
    }catch(InterruptedException e){  }
    //结束两个线程
    trHigh.normalStop();
    trLow.normalStop();
    //等待它们真正结束
    try{
      trHigh.join();
      trLow.join();
    }catch(InterruptedException e){  }
    //输出两个线程的循环次数
    System.out.println("trHigh 的循环次数为："+trHigh.getClick());
    System.out.println("trLow 的循环次数为："+trLow.getClick());
  }
}
```

在笔者的机器上，程序的输出结果为：

```
trHigh 的循环次数为：2031959251
trLow 的循环次数为：53200783
```

结果表明，优先级高的线程获得了更多的 CPU 运行时间。

8.4　线程的通信与协调

和进程一样，多线程的程序也要考虑各个线程之间的协调和配合。特别是当线程要共享资源时，就必须考虑线程之间的互斥、同步问题。如前面的例 8.1，count 变量的值之所以会出现错误，是因为没有考虑线程间的互斥问题。

多线程的程序，如果编写不当，还有可能发生死锁。关于互斥、同步、死锁、临界区这些概念，本节只做一个简单的介绍。详细的资料请参阅《操作系统》教程。

- 互斥：当多个线程需要访问同一资源，而这一资源在某一时刻只允许一个线程访问，那么这些线程就是互斥的。例如，线程 A 需要读取变量 comm，而线程 B 会给变量 comm 赋值，则 A 和 B 是互斥的。
- 同步：多个线程需要访问同一资源，而且需要相互配合才能正常工作，那么这些线程运行时就是一种同步关系。例如，线程 A 需要从缓冲区中读取数据，如果缓冲区为空则无法读取；而线程 B 会往缓冲区中写入数据，如果缓冲区已满则无法写入。那么 A 和 B 是同步线程。
- 临界区：为了实现线程间的互斥和同步，需要将共享资源放入一个区域，该区域一次只允许一个线程进入，该区域被称为临界资源。线程在访问共享资源前需要进行检查，看自己能否对该资源访问。如果有权访问，还需要阻止其他线程进入该区域。该代码段就是临界区。
- 死锁：若有多个线程相互等待其他线程释放资源，且所有线程都不释放自己所占有的资源，从而导致相关线程处于永远等待的状态，这种现象称为线程的死锁。

8.4.1　线程的互斥

为了解决进程间的互斥、同步，必须要使用信号量，而且信号量的设置必须使用 PV 原语。而在 Java 中，信号量需要用户自己管理，系统只提供了起到 PV 原语作用的三个方法以及一个关键字：

- public final void wait()：告知被调用的线程放弃管程进入睡眠，直到其他线程进入相同管程并且调用了 notify()。
- public final void notify()：恢复相同对象中第一个调用 wait()的线程。
- public final void notifyAll()：恢复相同对象中所有调用 wait()的线程。具有最高优先级的线程最先运行。

上面三个方法是 object 类的成员方法，由于该类是所有类的基类，所以在任何类中，可以直接使用这三个方法，无须用对象名.方法名()的格式。

wait()是将本线程转入阻塞状态，它和 sleep()不同，它会暂时释放占用的资源管程，wait(int mill)允许用户指定阻塞的时间。notify()是唤醒某个在管程队列中排队等候的线程，notifyAll()则是唤醒所有的阻塞线程。相对而言，后者更为安全一些。

除了这三个方法之外，还有一个关键字也经常要被用到：synchronized。

synchronized 关键字则用来标志被同步使用的资源。这里的资源既可以是数据，也可以是方法，甚至是一段代码。凡是被 synchronized 修饰的资源，系统都会为它分配一个管程，这样就能保证在某一时间内，只有一个线程对象在享有这一资源。这有点类似于街头的电话亭，当某人进去之后，可以从里面将其锁上。当另一个线程试图调用同一对象上的 Syncronized 方法时，它无法打开电话亭的门，因此它将停止运行。所以 synchronized 也被称为“对象锁”。而且，上面提到的三个方法，都只能使用在由 synchronized 控制的代码块中。

synchronized 的使用形式有两种，一种是保护整个方法：

```
访问类型 synchronized 返回值 方法名（[参数表]）{ …… }
```

另外一种是保护某个指定的对象以及随后的代码块：

```
synchronized(对象名){ …… }
```

要实现线程的互斥，需要以下几个步骤：

- 设置一个各个线程共享的信号量，值为 true 或者 false。
- 线程需要访问共享资源前，先检测信号量的值。如果不可用，则调用 wait()转入等待状态。
- 如果可用，则改变信号量的状态，不让其他线程进入。
- 访问完共享资源后，再修改信号量的状态，允许其他线程进入。
- 调用 notify()或 notifyAll()，唤醒其他等待的线程。

下面这个例子演示了如何实现三个线程之间的互斥。

【例 8.5】线程互斥示例。

//----------文件名 mutexThread.java，程序编号 8.9----------

```
public class  mutexThread extends Thread{  //定义 Thread 类的子类
  private static int count=0;               //这是静态变量
  private static boolean flag = true;       //信号量,用于线程间的互斥
  //这个 run()方法被 synchronized 所控制
  public  synchronized void run(){
     int i;
     for(i=0;i<100;i++){
       if (!flag)        //检测信号量是否可用
         try{
            wait();      //不允许进入临界区，等待
         }catch (InterruptedException e) { }
       flag = false;     //修改信号量，阻止其他线程进入
       count = count+1;  //访问共享资源
       flag = true;      //修改信号量，允许其他线程进入
       notifyAll();      //唤醒其他等待的线程
       System.out.println("My name is "+getName()+" count= "+count);
       try{
         sleep(10);      //让其他线程有机会获取 CPU
       }catch (InterruptedException e) { }
     }
  }
```

```
  public mutexThread(String name){
    super(name);
  }
}
```

下面再写一个程序测试它的运行情况：

//----------文件名 demoMutex.java，程序编号 8.10----------

```
public class demoMutex{
  public static void main(String argv[]){
    mutexThread trFirst,trSecond,trThird;
    //创建 3 个线程对象
    trFirst=new mutexThread("First Thread");
    trSecond=new mutexThread("Second Thread");
    trThird=new mutexThread("Third Thread");
    //启动这 3 个线程
    trFirst.start();
    trSecond.start();
    trThird.start();
  }
}
```

程序输出结果的前几行如下：

```
My name is First Thread count= 1
My name is Second Thread count= 2
My name is Third Thread count= 3
My name is First Thread count= 4
My name is Second Thread count= 5
My name is Third Thread count= 6
……
```

无论 run()方法循环多少次，也无论有多少个线程来访问 count 变量，count 的值都不会像在 8.3.1 小节中提到的那样出现错误。

不过，读者可能还会观测到这样的输出的结果：

```
……
My name is First Thread count= 211
My name is First Thread count= 212
My name is Second Thread count= 213
……
```

这是因为可能有某个线程连续获得 CPU 资源而连续运行。如果要避免这种情况，需要协调各个线程间的同步。

8.4.2　线程的同步

在某些情况下，两个（或者多个）线程需要严格交替地运行。比如，有一个存储单元，一个线程向这个存储单元中写入数据，另外一个线程从这个存储单元中取出数据。这就要

求这两个线程必须要严格交替的运行。要实现这一点，需要对线程进行同步控制。

同步控制的基本思路和互斥是一样的，也是通过信号量配合 wait()、notify()方法进行。不同的是，两个需要同步的线程会根据信号量的值，判断自己是否能进入临界区。比如，一个线程只有当信号量为真时才进入，而另外一个线程只有当信号量为假时才进入。而且只需要用 notify()通知另外一个等待线程就可以了。

下面写一个例子来演示线程的同步。由于这两个线程执行的任务不同，所以需要由两个不同的线程类来创建。前面实现线程间通信的时候，都是采用静态成员变量作为信号量，这里由于是不同的线程类，无法直接使用这种方式。一种容易想到的方法是设计一个公共类，信号量和共享资源都以静态成员变量的形式存在于类中。这样无论哪个线程对象，访问的都是同一个信号量和共享资源。这么做最为简单，但是不大符合 OOP 对数据封装的要求。8.4.4 小节将会采用另外一种解决办法。

这里面临的另外一个棘手的问题是：由于两个线程对象分属于不同线程类，而 notify()只能通知本线程类的其他对象，所以需要用“对象名.notify()”的形式，唤醒指定的其他线程类创建的线程对象。

【例 8.6】线程同步示例。

//----------文件名 commSource.java，程序编号 8.11----------

```
public class commSource{
  static boolean flag = true;
  static int data;
}
```

//----------文件名 setDataThread.java，程序编号 8.12----------

```
public class  setDataThread extends Thread{
  private readDataThread otherThread=null; //存储另外一个线程对象
  public  void run(){
     for(int i=0;i<100;i++){
         if (!commSource.flag)
           try{
              synchronized(this) {   //锁定当前对象
                 wait();             //阻塞自己
              }
           }catch (InterruptedException e) {    }
         commSource.flag = false;    //重新设置标志
         commSource.data = (int)(Math.random()*1000);
         System.out.println("设置数据："+commSource.data);
         synchronized(otherThread) {//锁定另外一个线程对象
             otherThread.notify();  //唤醒另外一个线程对象
         }
    }
  }
  public void setOtherThread(readDataThread rt){
    otherThread = rt;               //存储另外一个对象
 }
}
```

//----------文件名 readDataThread.java，程序编号 8.13----------

```
public class  readDataThread extends Thread{
  private setDataThread otherThread=null;
  public  void run(){
     for(int i=0;i<100;i++){
         if (commSource.flag)
           try{
             synchronized(this) {
                 wait();
             }
           }catch (InterruptedException e) {   }
         commSource.flag = true;
         System.out.println("获得数据："+commSource.data);
         synchronized(otherThread) {
             otherThread.notify();
         }
     }
  }
  public void setOtherThread(setDataThread st){
      otherThread = st;
  }
}
```

//----------文件名 demoSynchrony.java，程序编号 8.14----------

```
public class demoSynchrony{
  public static void main(String argv[]){
    setDataThread setTr;
    readDataThread readTr;
    readTr=new readDataThread();
    setTr=new setDataThread();
    readTr.setOtherThread(setTr);   //将其他对象传递进去
    setTr.setOtherThread(readTr);
    readTr.start();
    setTr.start();
  }
}
```

程序运行的部分结果如下：

```
设置数据：326
获得数据：326
设置数据：928
获得数据：928
设置数据：866
获得数据：866
设置数据：893
获得数据：893
设置数据：629
获得数据：629
设置数据：211
```

```
获得数据：211
……
```

表明两个线程是严格交替运行的。

8.4.3 暂停、恢复和停止线程

在某些情况下，一个线程可能需要去暂停、恢复和终止另外一个线程。在 JDK1.2 以前的版本中，实现这些功能的方法分别是 suspend()、resume()和 stop()。但从 JDK1.2 以后，这些方法都已经被丢弃，原因是它们可能会引起严重的系统故障。

Thread 类的 suspend()方法不会释放线程所占用的资源。如果该线程在某处挂起，其他的等待这些资源的线程可能死锁。

Thread 类的 resume()方法本身并不会引起问题，但它不能离开 suspend()方法而独立使用。

Thread 类的 stop()方法同样已被弃用。这是因为该方法可能导致严重的系统故障。设想一个线程正在写一个精密的重要的数据结构，且仅完成一小部分，如果该线程在此刻终止，则数据结构可能会停留在崩溃状态。

因为在 JDK1.5 中不允许使用 suspend()，resume()和 stop()方法来控制线程，读者也许会想：那就没有办法来停止、恢复和结束线程。事实并非如此，只要程序员在 run()方法中定期检查某些信号量，就可以判定线程是否应该被挂起、恢复或终止它自己的执行。

其实，程序 8.7 中的那个 run()方法就是通过检测 running 变量来判断自己是否应该结束。下面把这个程序改动一下，让它具备挂起、恢复和终止的功能。这需要用到 wait()和 notify()方法。

【例 8.7】 自己编写线程的暂停、恢复和停止方法。

//----------文件名 enhanceThread.java，程序编号 8.15----------

```
public class enhanceThread extends Thread{
  public static final int STOP = 1;
  public static final int RUNNING = 2;
  public static final int SUSPEND = 3;
  private int state = STOP;
  public synchronized void run(){
    int cnt = 0;
    while(state!=STOP){ //无限循环
      if(state==SUSPEND){
        try{
          wait();
        }catch(InterruptedException e) {   }
      }
      ++cnt;
      System.out.println("线程正在运行:"+cnt);
      try{
        sleep(100);     //让其他线程有机会获取 CPU
      }catch (InterruptedException e) { }
    }
```

```
  }
  //终止线程运行
  public void normalStop(){
     state = STOP;
  }
  //将线程挂起
  public void normalSuspend(){
     state = SUSPEND;
  }
  //恢复线程运行
  public synchronized void normalResume(){
    state = RUNNING;
    notify();
  }
  public enhanceThread(){
    state = RUNNING;
  }
}
```

//----------文件名 demoEnhanceThread.java，程序编号 8.16----------

```
public class demoEnhanceThread{
  public static void main(String argv[]){
    enhanceThread tr = new enhanceThread();
    System.out.println("启动线程");
    tr.start();
    try{
      Thread.sleep(1000);
      System.out.println("将线程挂起");
      tr.normalSuspend();
      Thread.sleep(1000);
      System.out.println("恢复线程运行");
      tr.normalResume();
      Thread.sleep(1000);
      System.out.println("终止线程运行");
      tr.normalStop();
    }catch(InterruptedException e){ }
  }
}
```

运行程序 8.16，输出结果如下：

```
启动线程
线程正在运行:1
线程正在运行:2
线程正在运行:3
线程正在运行:4
线程正在运行:5
线程正在运行:6
线程正在运行:7
线程正在运行:8
线程正在运行:9
```

```
线程正在运行:10
将线程挂起
恢复线程运行
线程正在运行:11
线程正在运行:12
线程正在运行:13
线程正在运行:14
线程正在运行:15
线程正在运行:16
线程正在运行:17
线程正在运行:18
线程正在运行:19
线程正在运行:20
终止线程运行
```

8.4.4 生产者-消费者问题实例

生产者-消费者问题（Producer_consumer）是操作系统中一个著名的进程同步问题。它一般是指：有一群生产者进程在生产产品，并将此产品提供给消费者进程去消费。为使生产者进程和消费者进程能并发执行，在它们之间设置一个缓冲区，生产者进程可将它所生产的产品放入一个缓冲区中，消费者进程可从一个缓冲区取得一个产品消费。尽管所有的生产者进程和消费者进程都是以异步的方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品，也不允许生产者进程向一个已装有消息，但尚未被取走产品的缓冲区投放产品。这里将“进程”换成“线程”，问题仍然成立。下面要做的事情就是用线程来模拟这一过程。

其实在 8.4.2 小节的例 8.6 中，所演示的线程同步就是这个问题的一个简单特例：只有一个消费者和一个生产者，缓冲区的大小为 1。不过例 8.6 的设计上有一点问题，不符合 OOP 的原则，而且控制 wait()和 notify()时也过于麻烦。

这里对它进行改进。笔者采用的方法是设计一个公共类，并用这个类创建一个对象，信号量和共享资源都以静态成员变量的形式存在于该对象中。在创建线程对象时，将这个公共对象传递进去，作为线程对象的私有数据。这样无论哪个线程对象，访问的都是同一个信号量和共享资源。

同时，将生产方法和消费方法都封装在这个公共类中，这样就避免了使用形如“对象名.notify()”这样的麻烦。

【例 8.8】 生产者-消费者实例。

//----------文件名 common.java，程序编号 8.17----------

```
public class common{           //公共线程类
   private int production[];
   private int count;          //产品的实际数目
   private int BUFFERSIZE = 6; //缓冲区的大小

   public common(){
     production = new int[BUFFERSIZE];
```

```
      count = 0;
   }
   //从缓冲区中取数据
   public synchronized int get(){
      int result;
      //循环检测缓冲区是否可用
      while (count<=0)
         try{
            wait();
         }catch(InterruptedException e) { }
      result = production[--count];
      notifyAll();
      return result;
   }
   //向缓冲区中写数据
   public synchronized void put(int newproduct){
      //循环检测缓冲区是否可用
      while (count>=BUFFERSIZE)
         try{
            wait();
         }catch(InterruptedException e) { }
      production[count++]=newproduct;
      notifyAll();
   }
}
```

由于缓冲区是大于 1 的，同时会有多个生产线程或是消费线程等待进入，而且它也允许连续多个生产者线程或是消费者线程进入。所以这里的信号量不是一个 boolean 类型，而是一个介于[0，BUFFERSIZE]之间的整型数。

注意：它的检测语句是：

```
while (count>=BUFFERSIZE)
   try{
      wait();
   }catch(InterruptedException e) { }
```

而前面所有程序中，此处都是用的 if。因为可能出现这样的情况：某生产者线程检测时，count 值已经等于 BUFFERSIZE，它被阻塞在此处。然后一个消费者线程进入，将 count 值减 1，然后再调用 notifyAll()唤醒这个线程。而与此同时，另外一个生产者线程已经抢先进入，再次把 count 的值加 1。如果本线程不再检测 count 值而直接进入，将导致下标越界的错误。

//----------文件名 consumer.java，程序编号 8.18----------

```
public class consumer extends Thread{  //消费者线程类
   private common comm;
   public consumer (common thiscomm){
      comm=thiscomm;
   }
   public void run(){
```

```
        int i,production;
        for(i=1;i<=20;i++){
          production=comm.get();
          System.out.println("得到的数据是："+production);
          try{
            sleep(10);
          }catch (InterruptedException e) { }
        }
    }
}
```

//----------文件名 producer.java，程序编号 8.19----------

```
public class producer extends Thread{  //生产者线程类
    private common comm;
    public producer(common thiscomm){
        comm=thiscomm;
    }
    public synchronized void run(){
        int i;
        for(i=1;i<=10;i++){
          comm.put(i);
          System.out.println("生产的数据是："+i);
          try{
            sleep(10);
          }catch (InterruptedException e) { }
        }
    }
}
```

//----------文件名 producer_consumer.java，程序编号 8.20----------

```
public class producer_consumer{    //演示生产者-消费者线程
    public static void main(String argv[]){
        common comm=new common();
        //创建 2 个生产者和 1 个消费者线程
        producer ptr1=new producer(comm);
        producer ptr2=new producer(comm);
        consumer ctr=new consumer(comm);
        ptr1.start();
        ptr2.start();
        ctr.start();
    }
}
```

程序某次运行结果如下：

```
生产的数据是：1
生产的数据是：1
得到的数据是：1
生产的数据是：2
生产的数据是：2
```

```
得到的数据是：2
生产的数据是：3
生产的数据是：3
得到的数据是：3
生产的数据是：4
生产的数据是：4
得到的数据是：4
生产的数据是：5
生产的数据是：5
得到的数据是：5
生产的数据是：6
生产的数据是：6
得到的数据是：6
生产的数据是：7
得到的数据是：6
生产的数据是：7
得到的数据是：7
生产的数据是：8
得到的数据是：7
得到的数据是：8
生产的数据是：9
生产的数据是：8
得到的数据是：9
生产的数据是：10
得到的数据是：8
生产的数据是：9
得到的数据是：10
生产的数据是：10
得到的数据是：9
得到的数据是：10
得到的数据是：5
得到的数据是：4
得到的数据是：3
得到的数据是：2
得到的数据是：1
```

结果表明，该程序已经很好地解决了生产者线程和消费者线程间的同步问题。

8.5　本 章 小 结

本章介绍了使用多线程编程的一些基础知识。其中包括如何创建自己的线程、如何对线程进行控制、如何进行线程间的通信和协调。其中，线程的同步是最难掌握的部分，需要程序员花费大量的时间和精力进行调试。由于多线程的编制比单线程的编制要困难得多，所以，在什么情况下使用多线程是需要仔细斟酌的。而且还需要注意一点：如果程序创建了太多的线程，这反而会减弱程序的性能。因为线程间的切换是需要开销的。如果线程太多，更多的 CPU 时间会用于上下文转换而不是用来执行程序。